Processing Text and Data

Introduction

LiveCode has first-class text and data processing capabilities. LiveCode's unique chunk expressions – the ability to refer to text using English-like statements like "word 3 to 5 of myVariable", combined with other powerful features which include regular expressions, XML processing, associative arrays, data encoding and decoding functions and compression and encryption algorithms – make it easy and simple to process text and data of any complexity. This chapter is a reference guide - go here for more resources.

The section *Processing Text and Data* in the *Sample Scripts* within the product documentation contains additional code examples.

Using Chunk Expressions

Chunk expressions are the primary method of working with text in LiveCode. A chunk is an English-like way of describing an exact portion of text. You can use chunks both to retrieve a portion of text, and to edit text. This topic defines the types of chunks you can address and describes the syntax for specifying them.

Types of Chunks

The common types of chunks are the **character**, **word**, **line**, or **item**. An item can be delimited by any character you specify. In addition, the **token** chunk is useful when parsing script data.

Here is an example of a chunk expression using the **word** chunk:

```
put word 1 to 3 of field "text" into myVariable
```

Using Chunks with Containers

You can use a chunk of a *container* anywhere you use an entire container. For example, you can use the **add**command to add a number to a line of a field:

```
add 1 to word 3 of field "Numbers"
```

You can also use chunk expressions to replace (using the **put** command) or remove (using the **delete**command) any portion of a container.

Using Chunks with Properties

You *can* use chunk expressions to *read* portions of a *property* (such as the **script** property). However, since you change a property with the **set** command rather than the **put** command, you *can't* use a chunk expression to *change* a part of a property's value. Instead, **put** the property value into a variable, use the chunk expression to change the variable, then **set** the property to the variable's contents. The following example shows how to change the third line of an object's **script** property:

```
put the script of me into tempScript
put "-- Last changed by Jane" into line 3 of tempScript
set the script of me to tempScript
```

The basic chunk types

There are four chunk types that are probably the most useful for basic text processing: **character**, **segment**, **item** and **line**.

The Character Chunk

A **character** is a single character, which may be a letter, digit, punctuation mark, or control character.

You can use the abbreviation **char** as a synonym for **character** in a chunk expression.

> **Important:** The character chunk corresponds to the notion of grapheme in the Unicode standard.

The Segment Chunk

A **segment** is a string of characters delimited by **space**, **tab**, or **return** characters or enclosed by double quotes.

> *Note:* **Word** is a synonym of segment.

The Item Chunk and the itemDelimiter Property

By default, an **item** is a string of characters delimited by commas.

Items are delimited by a string specified in the **itemDelimiter** property. You can change the default *comma* to create your own chunk type by setting the **itemDelimiter** property to any string.

The Line Chunk and the lineDelimiter Property

By default, a **line** is a string of characters delimited by the **return** character.

Lines are delimited by string in the **lineDelimiter** property. By default, the **lineDelimiter** is set to **return**, but you can create your own chunk type by setting the **lineDelimiter** property to any string.

Other chunks

There are also some chunks which are useful for more advanced text processing. These are:

- paragraph
- sentence
- trueWord
- codepoint
- codeunit
- byte

A **token** is a string of characters delimited by certain punctuation marks. The **token** chunk is useful in parsing LiveCode statements, and is generally used only for analyzing scripts.

The **sentence** and **trueWord** chunk expressions facilitate the processing of text, taking into account the different character sets and conventions used by various languages. They use the ICU library, which uses a large database of rules for its boundary analysis, to determine sentence and word breaks.

The **paragraph** chunk is currently identical to the existing **line** chunk, however in the future **paragraph**chunks will also be delimited by the Unicode paragraph separator.

The **codepoint** chunk type allows access to the sequence of Unicode codepoints which make up the string. The **codeunit** chunk type allows direct access to the UTF-16 code-units which notionally make up the internal storage of strings.
The **codeunit** and **codepoint** chunk are the same if a string only contains unicode codepoints from the Basic Multilingual Plane.

The **byte** chunk is an 8-bit unit and should be used when processing binary data.

For more information on the above chunk types, please consult the LiveCode Dictionary.

Specifying a Chunk

The simplest chunk expression specifies a single chunk of any type. The following statements all include valid chunk expressions:

```
get char 2 of "ABC" *-- yields "B"*
get segment 4 of "This is a test" *-- yields "test"*
get line 7 of myTestData
put "A" into char 2 of myVariable
```

You can also use the ordinal numbers **first**, **last**, **middle**, **second**, **third**, **fourth**, **fifth**, **sixth**, **seventh**,**eighth**, **ninth**, and **tenth** to designate single chunks. The special ordinal **any** specifies a *random* chunk.

```
put "7" into last char of "1085" -- yields "1087"
```

Negative Indexes in Chunk Expressions

To count *backwards* from the end of the value instead of forward from the beginning, specify a *negative*number. For example, the number -1 specifies the last chunk of the specified type, -2 specifies the next-to-last chunk, and so forth. The following statements all include valid chunk expressions:

```
get item -1 of "feather, ball, cap" -- yields "cap"
get char -3 of "ABCD" -- yields "B"
```

Complex Chunk Expressions

More complex chunk expressions can be constructed by specifying a chunk within another chunk. For example, the chunk expression `segment 4 of line 250` specifies the fourth segment of line 250.

When combining chunks of different types to construct a complex chunk expression, you must specify the chunk types in order. The following statements all include valid chunk expressions:

```
char 7 of segment 3 of myValue
segment 9 of item 2 of myValue
last char of segment 8 of line 4 of myValue
```

These, however, are not valid chunk expressions:

```
segment 8 of char 2 of myValue --chars can't contain segments

item 9 of first segment of myValue --segments can't contain items

line 3 of last item of myValue --items can't contain lines
```

The full hierarchy is as follows:

```
paragraph > sentence > line > item > segment >
    trueWord > token > character > codepoint > codeunit > byte
```

Using Parentheses in Chunk Expressions

You use parentheses in chunk expressions for the same reasons they're used in arithmetic:

To make a complex expression clearer.

To change the order in which the parts of the expression are evaluated.

For example, consider the following statement:

```
put item 2 of segment 3 of "a,b,c i,j,k x,y,z" -- BAD
```

The desired result is "y", the second item in the third segment. But the statement above causes an execution error, because it asks for an item of a segment, and segments can't contain items. You can obtain the desired result by using parentheses to change the order of evaluation:

```
put item 2 of (segment 3 of "a,b,c i,j,k x,y,z") -- good
```

In the example above, LiveCode gets the third segment first, then gets the second item in that segment. By adding parentheses around (segment 3 of "a,b,c i,j,k x,y,z"), you force LiveCode to evaluate that part of the chunk expression first. The value of the expression in parentheses is "x,y,z", and item 2 of "x,y,z" is "y".

As with arithmetic expressions, the parts of a chunk expression that are in parentheses are evaluated first. If parentheses are nested, the chunk expression within the innermost set of parentheses is evaluated first. The part that is enclosed in parentheses must be a valid chunk expression, as well as being part of a larger chunk expression:

```
put line 2 of segment 1 to 15 of myValue -- won't work
put line 2 of segment (1 to 15 of myValue) -- won't work
put line 2 of segment 1 to 15 (of myValue) -- won't work
put line 2 of (segment 1 to 15 of myValue) -- works!
```

The first of the above examples doesn't work for much the same reason as the previous example: segments can't contain lines. The second and third examples don't work because neither "1 to 15 of myValue" nor "of myValue" is a valid chunk expression. However, "segment 1 to 15 of myValue" is a valid chunk expression, so the last example works.

Nonexistent Chunks

If you request a chunk number that doesn't exist, the chunk expression evaluates to empty. For example, the expression `char 7 of "AB"` yields empty.

If you attempt to change a chunk that doesn't exist, what happens depends on what kind of chunk you specify:

Nonexistent character or segment:

Putting text into a character or segment that doesn't exist *appends* the text to the end of the container, without inserting any extra spaces.

Nonexistent item:

Putting text into an item that doesn't exist *adds* enough **itemDelimiter** characters to bring the specified item into existence.

Nonexistent line:

Putting text into a line that doesn't exist *adds* enough **return** characters to bring the specified line number into existence.

Specifying a Range

To specify a portion larger than a single chunk, you specify the beginning and end of the range. These are all valid chunk expressions:

```
get char 1 to 3 of "ABCD" -- yields "ABC"
get segment 2 to -1 of myValue -- second segment to last segment
put it into line 7 to 21 of myValue -- replaces
```

The start and end of the range must be specified as the same chunk type, and the beginning of the range must occur *earlier* in the value than the end. The following are not valid chunk expressions:

```
char 3 to 1 of myValue -- won't work
-- end cannot be greater than start

char -1 to -4 of myValue -- won't work
-- 4th from last comes before last
```

Important: When using negative numbers in a range, remember that numerically, -x comes after `-x+1`. For example, -1 is greater than -2, and -4 is greater than -7. The greater number must come **last** in order to create a valid range.

Counting the Number of segments, Lines or Items

The **number** function returns the number of chunks of a given type in a value. For example, to find out how many lines are in a variable, use an expression such as:

```
the number of lines in myVariable
```

You can also nest chunk expressions to find the number of chunks in a single chunk of a larger chunk type:

```
the number of chars of item 10 of myVariable
```

Comparing and Searching

LiveCode provides a number of ways of comparing and searching text. For most types of searching and comparing, you will find chunk expressions easy and convenient. However, if you have complex searching needs, you may prefer to use Regular Expressions, covered in the next section.

Checking if a Part is within a Whole

You use the **is in** operator to check if some text or data is within another piece of text or data. You can use the reverse **is not in** operator to check if text or data is not within another piece of text or data.

```
"A" is in "ABC" -- evaluates to true
"123" is in "13" -- evaluates to false
```

You can also use the **is in** operator to check whether some text or data is within a specified chunk of another container.

```
"A" is in item 1 of "A,B,C" -- evaluates to true
"123" is in segment 2 of "123 456 789" -- evaluates to false
```

Case Sensitivity

Comparisons in LiveCode are case insensitive by default (except for Regular Expressions, which have their own syntax for specifying whether or not a match should be case sensitive). To make a comparison case sensitive, set the **caseSensitive** property to true. For more details, see the *caseSensitive* property in the *LiveCode Dictionary*.

Checking if text is True, False, a Number, an Integer, a Point, a Rectangle, a Date or a Color

Use the **is a** operator for checking whether the user has entered data correctly and for validating parameters before sending them to a handler. The **is an** operator is equivalent to the **is a** operator.

A value **is a**:

- **boolean** if it is either true or false
- **integer** if it consists of digits (with an optional leading minus sign)
- **number** if it consists of digits, optional leading minus sign, optional decimal point, and optional "E" or "e" (scientific notation)
- **point** if it consists of two numbers separated by a comma
- **rect** if it consists of four numbers separated by commas
- **date** if it is in one of the formats produced by the date or time functions
- **color** if it is a valid color reference

The text you are checking can contain leading or trailing white space characters in all the types except boolean. For example:

```
" true" is true -- evaluates to false
```

The **is a** operator is the logical inverse of the **is not a** operator. When one is true, the other is false.

```
"1/16/98" is a date -- evaluates to true
1 is a boolean -- evaluates to false
45.4 is an integer -- evaluates to false
"red" is a color -- evaluates to true
```

Tip: To restrict a user to typing numbers in a field, use the following script

```
on keyDown pKey
    if pKey is a number then pass keyDown
end keyDown
```

The **keyDown** message will only be passed if the key the user pressed is a number. If you trap a **keyDown**message and don't pass it, the key will not be entered into the field. For more details, see the **keyDown**message in the *LiveCode Dictionary*.

Check if a segment, Item or Line Matches Exactly

The **is among** operator tells you whether a whole chunk exists exactly within in a larger container. For example, to find out whether the whole segment "free" is contained within a larger string, use the **is among**operator:

```
"free" is among the segments of "Live free or die" -- true
"free" is among the segments of "Unfree world" -- false
```

The second example evaluates to false because, although the string "free" is found in the value, it's a portion of a larger segment, not an entire segment.

Check if one String Starts or Ends With Another

To check if one string begins with or ends with another, use the **begins with** or **ends with** binary operators. For example:

```
“foobar” begins with “foo” -- true
“foobar” ends with “bar” -- true
line 5 of tList begins with "the"
```

Replacing Text

To replace one string with another, use the **replace** command. (If you want the search string to contain a regular expression, see the section on the *replaceText command* below instead.)

```
replace "A" with "N" in thisVariable -- changes A to N
```

To delete text using replace, replace a string with the empty constant.

```
replace return with empty in field 1 -- runs lines together
```

To preserve styling when replacing one string with another in a field, use the `preserving styles` clause. replace "foo" with "bar" in field "myStyledField" preserving styles -- replaces instances of "foo" with "bar" in the given field, -- "bar" will retain the styling of instances of "foo"

For more details, see the *replace command* and the *replace in field command* in the *LiveCode Dictionary*.

Retrieving the Position of a Matching Chunk

The **offset**, **itemOffset**, **tokenOffset**, **trueWordOffset**, **segmentOffset**, **lineOffset**, **sentenceOffset** and**paragraphOffset** functions can be used to locate the position chunks within a larger container. For example, this expression returns the character number where the letter "C" was found:

```
get offset("C","ABC") -- returns 3
```

To check if an item, line or word matches *exactly* using offset, set the **wholeMatches** property to true.

Chunks Summary

A chunk expression describes the location of a piece of text in a longer string.

Chunk expressions can describe **characters**, **items**, **tokens**, **trueWords**, **segments**, **lines**, **sentences** and **paragraphs** of text.

To count backward from the end of a string, use negative numbers. For example,

```
segment -2  -- indicates the second-to-last segment
```

You can combine chunk expressions to specify one chunk that is contained in another chunk, as in

```
segment 2 of line 3 of myVariable
```

For a range of chunks, specify the start and end points of the range, as in line 2 to 5 of myVariable

To check if a chunk is within another, use the **is in** operator. To check if a chunk **is a** specified type of data, use the **is a** operator. To check if a chunk starts or ends with another uses the **begins with** or ends with operators.

To check if a chunk is contained exactly within a string use the **is among** operator. To get an index specifying where a chunk can be found in a container, use the **\Offset** functions described above. To match only a complete chunk within a string, set the wholeMatches to true before using the offset functions.

Regular Expressions

Regular expressions allow you to check if a *pattern* is contained within a string. Use regular expressions when one of the search or comparison chunk expressions does not do what you need (see the section on *Comparing and Searching* above).

LiveCode supports searching for a pattern, replacing a pattern, or filtering the lines in a container depending on whether or not each line contains the pattern. Regular expressions use PERL compatible or "PCRE" syntax. Figure 52, below, shows the supported syntax. For more details on the supported syntax, see thePCRE manual

Searching using a Regular Expression

Use the **matchText** function to check whether a string contains a specified pattern.

```
matchText(string,regularExpression[,foundTextVarsList])
```

The *string* is any expression that evaluates to a string.

The *regularExpression* is any expression that evaluates to a regular expression.

The optional *foundTextVarsList* consists of one or more names of existing variables, separated by commas.

```
matchText("Goodbye","bye") -- returns true
matchText("Goodbye","^Good") -- also returns true
matchText(phoneNumber,"([0-9]+)-([0-9]+-[0-9]+)",areaCode,phone)
```

For more details on this function see the *matchText* function in the *LiveCode Dictionary*.

If you need to retrieve the positions of the matched substrings in the optional *foundTextVarsList*, use the *matchChunk* function instead of the *matchText* function. These functions are otherwise identical.

Replacing using a Regular Expression

Use the **replaceText** function to search for a regular expression and replace the portions that match. If you simply want to replace text without using a regular expression, see the **replace** command instead.

```
replaceText(stringToChange,matchExpression,replacementString)
```

The *stringToChange* is any expression that evaluates to a string.

The *matchExpression* is a regular expression.

The *replacementString* is any expression that evaluates to a string.

```
replaceText("malformed","mal","well")--returns "wellformed"
replaceText(field "Stats",return,comma)-- makes comma-delimited
```

For more details, see the *replaceText* function in the *LiveCode Dictionary*.

Regex	Rule	Example
[chars]	matches any one of the characters inside the brackets	A[BCD]E matches "ACE", but not "AFE" or "AB"
[^chars]	matches any single character that is not inside the brackets	[^ABC]D matches "FD" or "ZD", but not "AD" or "CD"
[char-char]	matches the range from the first char to the second char. The first char's ASCII value must be less than the second char's ASCII value	A[B-D] matches "AB" or "AC", but not "AG" [A-Z0-9] matches any alphanumeric character
.	matches any single character (except a linefeed)	A.C matches "ABC" or "AGC", but not "AC" or "ABDC"
^	matches the following character at the beginning of the string	^A matches "ABC" but not "CAB"
$	matches the preceding character at the end of a string	B$ matches "CAB" but not "BBC"
`*`	matches zero or more occurrences of the preceding character or pattern	ZA*B matches "ZB" or "ZAB" or "ZAAB", but not "ZXA" or "AB" [A-D]*G matches "AG" or "G" or "CAG", but not "AB"

Regex	Rule	Example
+	matches one or more occurrences of the preceding character or pattern	ZA+B matches "ZAB" or "ZAAB", but not "ZB" [A-D]+G matches "AG" or "CAG", but not "G" or "AB"
?	matches zero or one occurrences of the preceding character or pattern	ZA?B matches "ZB" or "ZAB", but not "ZAAB" [A-D]?G matches "AG" or "CAGZ", but not "G" or "AB"
**\	**	matches either the pattern before or the pattern after the \	.	A\	B matches "A" or "B" [ABC]\	[XYZ] matches "AY" or "CX", but not "AA" or "ZB".
****	Causes the following character to be matched literally, even if it has special meaning in a regular expression	A\.C matches "A.C", but not "A\.C" or "ABC"\\ matches "\"
any other character	matches itself	ABC matches "ABC"

Figure 50 – Regular Expression Syntax

Filtering using a Wildcard Expression

Use the **filter** command to remove lines in a container that either do, or do not match a specified wildcard expression. Wildcard expressions are similar to regular expressions.

```
filter container {with | without} wildcardExpression
```

The *container* is any expression that evaluates to a container.

The *wildcardExpression* is a pattern used to match certain lines.

```
filter myVariable with "A?2"
filter me without "\*[a-zA-Z]\*"
```

For more details, including the format of wildcard expressions, see the *filter command* in the *LiveCode Dictionary*.

International Text Support

All LiveCode's text processing capabilities extend seamlessly to international text. This includes the ability to render and edit Unicode text and convert between various encoding types.

What are Text Encodings?

Fundamentally computers use numbers to store information, converting those numbers to text to be displayed on the screen. A text encoding describes which number converts to a given character. There are many different encoding systems for different languages. Below is a table containing examples of some common encodings.

Encoding	Representation	Description
ASCII	Single byte – English	ASCII is a 7-bit encoding, using one byte per character. It includes the full Roman alphabet, Arabic numerals, Western punctuation and control characters. See http://en.wikipedia.org/wiki/ASCII for more information.
ISO8859	Single byte	ISO8859 is a collection of 10 encodings. They are all 8-bit, using one byte per character. Each shares the first 128 ASCII characters. The upper 80 characters change depending on the language to be displayed. For example ISO8859-1 is used in Western Europe, whereas ISO8859-5 is used for Cyrillic. **NB:** LiveCode only supports ISO8859-1. You should use Unicode to represent other languages, converting if necessary (see below).
Windows-1252	Single byte – English	This is a superset of ISO8859-1 which uses the remaining 48 characters not used in the ISO character set to display characters on Windows systems. For example curly quotes are contained within this range.
MacRoman	Single byte – English	MacRoman is a superset of ASCII. The first 128 characters are the same. The upper 128 characters are entirely rearranged and bear no relation to either Windows-1252 or ISO8859-1. However while many of the symbols are in different positions many are equivalent so it is possible to convert between the two.
UTF-16	Double byte – Any	UTF-16 typically uses two bytes per code point (character) to display text in all the world's languages (see *Introduction to Unicode*, below). UTF-16 will take more memory per character than a single-byte encoding and so is less efficient for displaying English.
UTF-8	Multi-byte - Any	UTF-8 is a multi-byte encoding. It has the advantage that ASCII is preserved. When displaying other languages, UTF-8 combines together multiple bytes to define each code point (character). The efficiency of UTF-8 depends on the language you are trying to display. If you are displaying Western European it will take (on average) 1.3 bytes per character, for Russian 2 bytes (equivalent to UTF-16) but for CJK 3-4 bytes per character.

Figure 51 – Common text encodings

What are scripts?

A script is a way of writing a language. It takes the encoding of a language and combines it with its alphabet to render it on screen as a sequence of glyphs. The same language can sometimes be written with more than one script (common among languages in India). Scripts can often be used to write more than one language (common among European languages).

Scripts can be grouped together into four approximate classes. The "small" script class contains a small alphabet with a small set of glyphs to represent each single character. The "large" script class contains a large alphabet and with a larger set of glyphs. The "contextual" script class contains characters that can change appearance depending on their context. And finally the "complex" script class contains characters that are a complex function of the context of the character – there isn't a 1 to 1 mapping between code point and glyph.

Roman	Small script	The Roman encoding has relatively few distinct characters. Each character has a single way of being written. It is written from left to right, top to bottom. Every character has a unique glyph. Characters do not join when written. For example: The quick brown fox.
Chinese	Large script	The Chinese encoding has a large number of distinct characters. Each character has a single way of being written.
Greek	Contextual script	Every character except sigma has a unique glyph. Sigma changes depending on whether it is at the end of a word or not. Characters do not join when written. The text runs left to right, top to bottom. For example: δῖος Ἀχιλλεύς
Arabic	Contextual script	The glyph chosen is dependent on its position in a word. All characters have initial, medial and terminal glyphs. This results in a calligraphic (joined up) style of display. The text runs right to left, top to bottom display. For example: العربية
Devanagari	Complex script	In this script there is no direct mapping from character to glyph. Sequences of glyphs combine depending on their context. The text runs from left to right, top to bottom.

Figure 52 – Common scripts

Introduction to Unicode

Traditionally, computer systems have stored text as 8-bit bytes, with each byte representing a single character (for example, the letter 'A' might be stored as 65). This has the advantage of being very simple and space efficient whilst providing enough (256) different values to represent all the symbols that might be provided on a typewriter. The flaw in this scheme becomes obvious fairly quickly: there are far more than 256 different characters in use in all the writing systems of the world, especially when East Asian ideographic languages are considered. But, in the pre-internet days, this was not a big problem.

LiveCode, as a product first created before the rise of the internet, also adopted the 8-bit character sets of the platforms it ran on (which also meant that each platform used a different character set: MacRoman on Apple devices, CP1252 on Windows and ISO-8859-1 on Linux and Solaris). LiveCode terms these character encodings "native" encodings.

In order to overcome the limitations of 8-bit character sets, the Unicode Consortium was formed. This group aims to assign a unique numerical value ("codepoint") to each symbol used in every written language in use (and in a number that are no longer used!). Unfortunately, this means that a single byte cannot represent any possible character.

The solution to this is to use multiple bytes to encode Unicode characters and there are a number of schemes for doing so. Some of these schemes can be quite complex, requiring a varying number of bytes for each character, depending on its codepoint.

LiveCode previously added support for the UTF-16 encoding for text stored in fields but this could be cumbersome to manipulate as the variable-length aspects of it were not handled transparently and it could only be used in limited contexts. Unicode could not be used in control names, directly in scripts or in many other places where it might be useful.

From LiveCode 7.0, the LiveCode engine has been able to handle Unicode text transparently throughout. The standard text manipulation operations work on Unicode text without any additional effort on your part; Unicode text can now be used to name controls, stacks and other objects; menus containing Unicode selections no longer require tags to be usable - anywhere text is used, Unicode should work.

Creating Unicode Apps

Creating stacks that support Unicode is no more difficult than creating any other stack but there are a few things that should be borne in mind when developing with Unicode. The most important of these is the difference between text and binary data - in previous versions of LiveCode, these could be used interchangeably; doing this with Unicode may not work as you expect (but it will continue to work for non- Unicode text).

When text is treated as binary data (i.e when it is written to a file, process, socket or other object outside of the LiveCode engine) it will lose its Unicode-ness: it will automatically be converted into the platform's 8-bit native character set and any Unicode characters that cannot be correctly represented will be converted into question mark '?' characters.

Similarly, treating binary data as text will interpret it as native text and won't support Unicode. To avoid this loss of data, text should be explicitly encoded into binary data and decoded from binary data at these boundaries - this is done using the textEncode and textDecode functions (or its equivalents, such as opening a file using a specific encoding).

Unfortunately, the correct text encoding depends on the other programs that will be processing your data and cannot be automatically detected by the LiveCode engine. If in doubt, UTF-8 is often a good choice as it is widely supported by a number of text processing tools and is sometimes considered to be the "default" Unicode encoding.

Things to look out for

- When dealing with binary data, you should use the byte chunk expression rather than char - char is intended for use with textual data and represents a single graphical character rather than an 8-bit unit.
- Try to avoid hard-coding assumptions based on your native language - the formatting of numbers or the correct direction for text layout, for example. LiveCode provides utilities to assist you with this.
- Regardless of visual direction, text in LiveCode is always in logical order - word 1 is always the first word; it does not depend on whether it appears at the left or the right.
- Even English text can contain Unicode characters - curly quotation marks, long and short dashes, accents on loanwords, currency symbols...

Using Arrays

For an introduction to arrays, see the section on *Array Variables* in the chapter *Coding in LiveCode*.

When to Use Arrays

Each element in an array can be accessed in constant time. This compares favorably with other functions that become look up information by counting through a variable from the beginning (for example the offset functions). If you consider a problem that requires the computer to search through the data several times then if the computer has to start at the beginning of the variable, the search will get slower and slower as the search function gets further through the data.

Each element in an array can contain data of any length, making it easier to group together records that contain assorted lengths or delimiter characters.

Arrays can contain nested elements. This makes them ideal for representing complex data structures such as trees and XML data that would be hard to represent as a flat structure.

Each sub-array in an array can be accessed and operated on independently. This makes it possible to copy a sub-array to another array, get the keys of a sub-array, or pass a sub-array as a parameter in a function call.

LiveCode includes various functions for converting information to and from arrays, and for performing operations on the contents of arrays.

These characteristics make arrays useful for a number of data processing tasks, including tasks that involve processing or comparing large amounts of data. For example, arrays are ideal if you want to count the number of instances of a specific word in a piece of text. It would be possible to implement such a word count by iterating through each word and checking if it is present in a list of words, then adding a comma followed by a count to that list of words. Such a method is cumbersome to implement and as the list of words gets longer the routine will slow down because LiveCode has to search the list from the start with each new word.

Conversely, implementation with an array is simple. Because each element in an array can be named using a text string, we can create an element for each word and add to the element's contents. Not only is the code much shorter but it is also an order of magnitude faster.

```
on mouseUp
    --cycle through each word adding each instance to an array
    repeat for each word tWord in field "sample text"
        add 1 to tWordCount[tWord]
    end repeat
    -- combine the array into text
    combine tWordCount using return and comma
    answer tWordCount
end mouseUp
```

Text in field "Sample Text:	Resulting value of tWordCount:
Single Line – execute single line and short scripts	Global,4
Multiple Lines – execute multiple line scripts	Line,3
Global Properties – view and edit global properties	Lines,1
Global Variables – view and edit global variables	Multiple,2
	Properties,2
	Single,2
	Variables,2
	and,3
	edit,2
	execute,2
	scripts,2
	short,1
	view,2
	-,4

Figure 53 – Results of running word count script

Array Functions in LiveCode

The following is a list of all the syntax in LiveCode that works with arrays. For a full description of each one, see the corresponding entry in the LiveCode Dictionary.

Each of these functions can be used on subarrays within an array. Instead of referring to the array variable, refer to x[x]

- **add** adds a value to every element in an array where the element is a number

- **combine** converts an array into text, placing delimiters you specify between the elements -**customProperties** returns an array of the custom properties of an object
- **delete variable** remove an element from an array
- **divide** divides each element in an array where the element is a number. For example:

  divide tArray by 3

Contents of array:	Resulting value of tWordCount:
A = 1	0.333333
B = 2	0.666667
C = 3	1
D = 4	1.333333
E = 5	1.666667

Figure 54 – Results of running the divide command on an array

- **element** keyword is used in a repeat loop to loop through every element in an array
- **extents** finds the minimum and maximum row and column numbers of an array whose keys are integers
- **intersect** compares arrays, removing elements from one array if they have no corresponding element in the other
- **keys** returns a list of all the elements within an array
- **matrixMultiply** performs a matrix multiplication on two arrays whose elements are numbers and whose keys are sequential numbers
- **multiply** multiplies a value in every element in an array where the element is a number
- **properties** returns an array of the properties of an object
- **split** convert text to an array using delimiters that you define
- **sum** - returns the sum total of all the elements in an array where the element is a number
- **transpose** swaps the order of the keys in each element in an array whose elements are numbers and whose keys are sequential numbers
- **union** combines two arrays, eliminating duplicate elements

Encoding and Decoding

LiveCode includes a number of built-in functions for encoding and decoding data in a variety of popular formats.

Styled Text

LiveCode supports encoding and decoding styled text as HTML and RTF. This feature is useful when you want to adjust text styles programmatically, or import or export text with style information.

> **Important:** At present HTML conversion support only extends to the styles that the LiveCode field object is capable of rendering.

To convert the contents of a field to HTML compatible tags, use the **HTMLText** property. This property is documented in detail in the LiveCode Dictionary. You can also set this property to display styled text in a field.

> **Tip:** You can get and set the HTMLText property of a chunk within a field, allowing you to view or change text attributes on a section of the text.

For example, to set the text style of line 2 of a field to bold:

```
on mouseUp
    put the htmltext of line 2 of field "sample text" into tText`
    replace "\<p\>" with "\<p\>\<b\>" in tText
    replace "\</p\>" with "\</b\>\</p\>\>" in tText
    set the htmltext of line 2 of field "sample text" to tText`
end mouseUp
```

While this is not as simple as directly applying the style to the text using:

```
set the textStyle of line 2 of field "sample" to "bold"
```

It does allow you to search and replace text styles or perform multiple complex changes to the text based on pattern matching. Performing a series of changes on the HTMLText in a variable then setting the text of a field once can be quicker than updating the style repeatedly directly on the field.

Use the **HTML** keyword with the *Drag and Drop* features and the *Clipboard* features to perform conversion of data to and from HTML when exchanging data with other applications. For more information see the section on *Drag and Drop* in the *Programming a User Interface* guide.

Use the **RTFText** property and **RTF** keyword to work with the RTF format.

Use the **unicodeText** property and **Unicode** keyword to work with Unicode. For more information see the section on *International Text Support*, above.

URLs

To encode and decode URLs, use the **URLEncode** and **URLDecode** functions. The **URLEncode** function will make text safe to use with a URL – for example it will replace *space* with +. These functions are particularly useful if you are posting data to a web form using the **POST** command, using the **launch URL** command or sending email using the **revMail** function. For more information see the *LiveCode Dictionary*.

Text:	URL Encoded result:
"Jo Bloggs" \jo@blogs.com\	%22Jo+Bloggs%22+%3Cjo%40blogs.com%3E

Figure 55 – Results of encoding a URL

Binary Data – Base64 (for MIME Email Attachments and Http Transfers)

To encode and decode data in Base64 (e.g. to create an email attachment), use the **base64Encode** and **base64Decode** functions. These functions are useful anywhere you want to convert binary data to text data and back. For more information see the *LiveCode Dictionary*.

Binary Data – Arbitrary Types

Use the **binaryEncode** and **binaryDecode** functions to encode or decode binary data. These functions are documented in detail in the *LiveCode Dictionary*.

Character to Number conversion

To convert a character to and from its corresponding ASCII value use the **charToNum** and **numToChar**functions.

```
put charToNum("a") -- results in 97
```

To convert Unicode characters, set the **useUnicode** local property to true. For more information see the section on *International Text Support*, above.

Data Compression

To compress and decompress data using GZIP, use the **compress** and **decompress** functions.

The following routine asks the user to select a file, then creates a GZip compressed version with a ".gz" extension in the same directory as the original.

```
on mouseUp
    answer file "Select a file:"
    if it is empty then exit mouseUp
    put it & ".gz" into tFileCompressed
    put compress(URL ("binfile:" & it)) into URL ("binfile:" & tFileCompressed)
end mouseUp
```

Encryption

To encrypt or decrypt data use the **encrypt** and **decrypt** commands. These commands are documented in the LiveCode Dictionary.

Generating a Checksum

Use the **MD5Digest** to generate a digest of some data. Use this function later to determine if the data was changed or to check that transmission of information was complete.

Tip: In this example we save the MD5Digest of a field when the user opens it for editing. In the field script place:

```
on openField
    set the cDigest of me to md5Digest(the htmlText of me)
end openField
```

If the field is modified (including if a text style is changed anywhere) then a subsequent check of the MD5Digest will return a different result. In the following example we check this digest to determine whether or not to bring up a dialog alerting the user to save changes:

```
on closeStackRequest
    local tDigest
    put md5Digest(the htmlText of field "sample text") into tDigest
    if the cDigest of field "sample text" is not tDigest then
        answer "Save changes before closing?" with "No" or "Yes"
        if it is "Yes" then
            save this stack
        end if
    end if
end closeStackRequest
```

XML

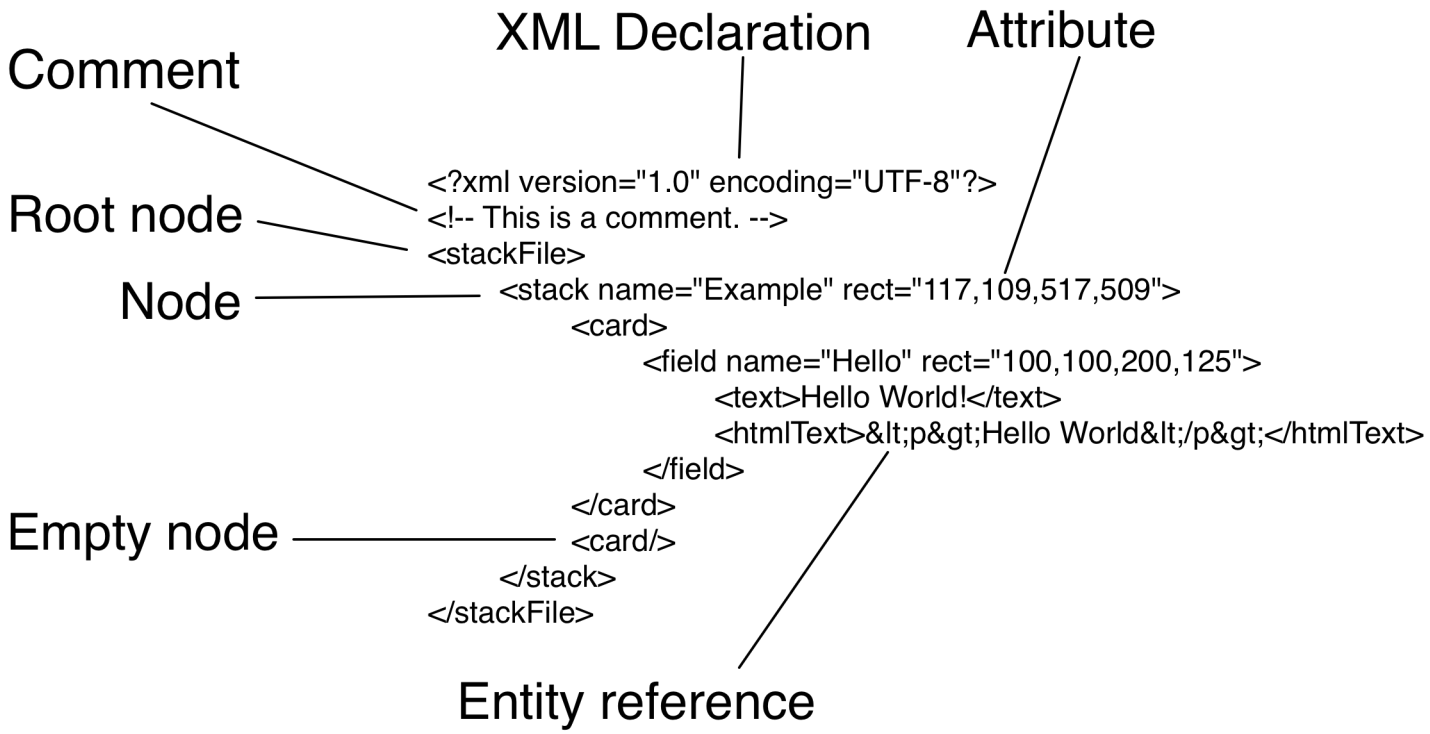
Extensible Markup Language, or XML, is a general-purpose language for exchanging structured data between different applications and across the Internet. It consists of text documents organized into a tree structure. It can generally be understood by both human and machine.

LiveCode includes comprehensive support for XML through its built-in XML library. Additionally, standards exist to support exchange of XML over a network connection (or "web services") – most notably through the XML-RPC and SOAP protocols. LiveCode includes a library for using XML-RPC and there are examples of using LiveCode to build SOAP applications available.

The XML Tree Structure

XML is simply a data tree. It must start with a root node, be well formed and nested. Tags may not overlap. For more information on XML see http://en.wikipedia.org/wiki/XML

Figure 56 below shows a typical XML tree. In this example we have represented a simple stack file as XML. The stack file has a single stack with two cards. On the first card there is a field named "Hello" with the contents "Hello World!". There is a second card, which is blank.



Root node	Root element, document element	The start of the XML document, which includes a declaration the file is XML, the version of XML in use and the text encoding
Comment		Comments can be placed anywhere in the tree. They start with \<! and end with ->. They must not contain double dashes --
Node	Element, tag	The items that makes up the content within an XML document
Attributes		Properties attributable to a given node. A node may have zero or more properties

Empty node	Empty element	A method of specifying that a node exists but is empty
Entity reference		A method of specifying special characters. XML includes support for &, \<, >, ‘ and ". Additional entities can be defined using a Document Type Definition or DTD.

When to use XML

XML has a number of advantages and disadvantages. It is predominantly useful when exchanging data between different applications or systems. However like any method of data storage or transfer it is not suitable for all types of application.

The advantages of XML are: it is text based making it more easily readable by humans as well as just machines; it is self describing; it is based on international standards and in widespread use with a large number of editors available for it; the hierarchical structure makes it suitable for representing many types of document; and it is platform independent.

The disadvantages are that: it is sometimes less efficient than binary or even other forms of text representations of data; for simple applications it is more complicated than may strictly be necessary; and the hierarchical model may not be suitable for all data types.

You may decide that using XML is the best solution for your particular data storage or transmission requirements. Or you may be working on a project with others where using XML or a web service based on it is a requirement. However in many cases a binary format or database will be more appropriate. You should give consideration to the method you intend to use as early as possible in the design of your application.

Methods for Handling XML in LiveCode

LiveCode includes a comprehensive XML library for working with XML documents. Using the XML library has the advantage that we include syntax and functions for performing the common operations on XML that you may need. However the disadvantage is that at present the library is implemented as an external command (included built-in to the LiveCode distribution) and thus does not benefit from native LiveCode-engine syntax. If you have simple XML processing requirements you may prefer to use LiveCode’s built in chunk expression support to do the parsing, matching or construction of XML. For more information see the section on *Using Chunk Expressions*. However if you are working with complex XML then the library includes a comprehensive suite of features.

In addition to the XML library, LiveCode has a built-in script-based library for working with XML-RPC.

Tip: To see a list of commands for working with XML-RPC, filter the LiveCode Dictionary with the term XMLRPC.

A lesson demonstrating using the LiveCode XML library is available here

The XML Library: Loading, Displaying and Unloading XML

This section discusses using the XML library in detail.

Getting Started – Creating an XML Tree in Memory

In order to work with an XML document, you start by creating an XML tree of that document in memory. There are two functions **revXMLCreateTreeFromFile** and **revXMLCreateTree**. Use the former to load XML document from a file and create a tree in memory, use the latter to create an XML tree from another data source such as a variable, field or download.

```
revXMLCreateTree(XMLText, dontParseBadData, createTree, sendMessages)
revXMLCreateTreeFromFile(filePath, dontParseBadData, createTree, sendMessages)
```

In **revXMLCreateTree** the *XMLText*is the string containing the XML. In **revXMLCreateTreeFromFile** this parameter is replaced with the *filePath* – the file path to the XML document. Both functions return a single value – the ID of the tree that has been created.

> **Important:** Both functions require you to specify all the parameters. You must store the ID returned by these functions in order to access the XML tree later in your script.

The dont*ParseBadData* specifies whether or not to attempt to parse poorly formed XML. If this is set to true then bad data will be rejected and generate an error instead of constructing the tree in memory.

The *createTree* specifies whether to create a tree in memory or not. You will generally want this to be true, unless you are intending only to read in an XML file to determine whether or not it is properly structured.

The *sendMessages* specifies whether or not messages should be sent when parsing the XML document. Messages can be useful if you want to implement functionality such as a progress bar, progressively render or progressively process data from a large XML file as it is being parsed. If you set this to true, **revXMLStartTree** will be sent when the parsing starts, **revStartXMLNode** will be sent when a new node is encountered, **revEndXMLNode** will be sent when a node has been completed, **revStartXMLData** will be sent at the start of a new block of data and finally **revXMLEndTree** will be sent when processing is finished.

Retrieving information from an XML tree

Now that you have created your XML tree in memory (above) and stored the tree ID you can use the functions in this section to retrieve information from within the tree.

> **Important:** Any text you fetch using the LiveCode XML library will be in the encoding specified in the root node of the XML tree.
>
> **Note:** All the examples in this section assume that we have loaded the XML tree depicted in the figure below – XML Tree Representation of a Stack, above. We assume that you have loaded this tree using the `revXMLCreateTree` function described above, and that this function has returned a value of 1 as the ID of the tree.

Retrieving the Root Node

To retrieve the *root node* from your XML tree, use the **revXMLRootNode** function.

```
revXMLRootNode(treeID)
```

The *treeID* contains the ID of the XML tree you want to access. For example, using the following function with sample tree depicted above:

```
put revXMLRootNode(1) into tRootNode
```

Results in *tRootNode* containing: *stackFile*

Retrieving the First Child Element in a Node

To retrieve the first child element use **revXMLFirstChild**.

```
revXMLFirstChild(treeID,parentNode)
```

The *parentNode* contains the path to the node we want to retrieve the first child from. Nodes are referenced using a file-path like format with / used to denote the root and delimit nodes.

We can use this function on our sample XML as follows:

```
-- pass the *stackFile* result in retrieved in tRootNode
-- to the revXMLFirstChild function:
put revXMLFirstChild(1,tRootNode) into tFirstChild
-- EQUIVALENT to the following:
put revXMLFirstChild(1,"stackFile") into tFirstChild
```

This results in *tFirstChild* containing: */stackFile/stack*

Retrieving a list of Children in a Node

To retrieve a list of children of a node use **revXMLChildNames**.

```
revXMLChildNames(treeID, startNode, nameDelim, childName,includeChildCount)
```

The *nameDelim* is the delimiter that separates each name that is returned. To get a list of names, specify return.

The *childName* is the name of the type of children to list.

includeChildCount allows you to include the number of each child in square brackets next to the name.

We can use this function on our sample XML as follows:

```
put revXMLChildNames(1,"/stackFile/stack", return, "card", true) into tNamesList
```

This results in *tNamesList* containing:

card[1]

card[2]

Retrieving the Contents of the Children in a Node

To retrieve a list of children of a node including their contents, use **revXMLChildContents**.

```
revXMLChildContents(treeID,startNode,tagDelim,nodeDelim, includeChildCount,depth)
```

See above for an explanation of *treeID*, *startNode* and *tagDelim*.

The *nodeDelim* indicates the delimiter that separates the contents of the node from its name.

The *depth* specifies the number of generations of children to include. If you use –1 as the depth then all children are return.

Using this function on our example XML file as follows:

```
put revXMLChildContents(1, "/stackFile/stack", space, return, true, -1) into tContents
```

This results in *tContents* containing:

card[1]

field[1]

text[1] Hello World!

htmlText[1] \Hello World

card[2]

Retrieving the Number of Children in a Node

To retrieve the number of children of a node **revXMLNumberOfChildren**.

```
revXMLNumberOfChildren(treeID,startNode,childName,depth)
```

See above for an explanation of *treeID*, *startNode, childName* and *depth*.

Using this function on our example XML file as follows:

```
put revXMLNumberOfChildren(1, "/stackFile/stack", "card", -1)into tContents
```

This results in *tContents* containing: *2*

Retrieving the Parent of a Node

To retrieve a node's parent use the **revXMLParent** function.

```
revXMLParent(treeID,childNode)
```

See above for an explanation of *treeID* and *startNode*.

Using this function on our example XML file as follows:

```
put revXMLParent(1,"stackFile/stack") into tParent
```

Results in *tParent* containing: */stackFile*

Retrieving an Attributes from a Node

To retrieve an attribute from a node use **revXMLAttribute**.

```
revXMLAttribute(treeID,node,attributeName)
```

See above for an explanation of *treeID* and *node*.

The *attributeName* is the name of the attribute you want to retrieve the value for.

Using this function on our example XML file as follows:

```
put revXMLAttribute(1,"/stackFile/stack","rect") into tRect
```

This results in *tRect* containing: *117,109,517,509*

Retrieving all Attributes from a Node

To retrieve all attributes from a node use **revXMLAttributes**.

```
revXMLAttributes(treeID,node,valueDelim,attributeDelim)
```

See above for an explanation of *treeID* and *node*.

The *valueDelim* is delimiter that separates the attribute's name from its value.

The *attributeDelim* is delimiter that separates the attribute's name & value pair from each other.

Using this function on our example XML file as follows:

```
put revXMLAttributes(1, "/stackFile/stack/card/field", tab, return) into tFieldAttributes
```

This results in *tFieldAttributes* containing:

name Hello

rect 100,100,200,125

Retrieving the Contents of Attributes

To retrieve the contents of a specified attribute from a node and its children, use **revXMLAttributeValues**.

```
revXMLAttributeValues(treeID, startNode, childName, attributeName, delimiter, depth)
```

See above for an explanation of *treeID*, *startNode* and *depth*.

The *childName* is the name of the type of child to be searched. Leave this blank to include all types of children.

The *attributeName* is the name of the attribute to return the values for.

The *delimiter* is the delimiter to be used to separate the values returned.

Using this function on our example XML file as follows:

```
put revXMLAttributeValues(1, "/stackFile/", , "rect", return, -1) into tRectsList
```

This results in *tRectsList* containing:

117,109,517,509

100,100,200,125

Retrieving the Contents of a Node

To retrieve the contents of a specified node, use **revXMLNodeContents**.

```
revXMLNodeContents(treeID, node)
```

See above for an explanation of *treeID* and *node*.

Using this function on our example XML file as follows:

```
put revXMLNodeContents(1, "/stackFile/stack/card/field/htmlText") into tFieldContents
```

This results in *tFieldContents* containing:

\Hello World\

The entity references for the \< and > symbols have been translated into text in this result.

Retrieving Siblings

To retrieve the contents of the siblings of a node, use **revXMLNextSibling** and **revXMLPreviousSibling**.

```
revXMLNextSibling(treeID,siblingNode)
revXMLPreviousSibling(treeID,siblingNode)
```

The *siblingNode* is the path to the node to retrieve the siblings from.

Using this function on our example XML file as follows:

```
put revXMLPreviousSibling(1, "/stackFile/stack/card[2]") into tPrev
put revXMLNextSibling(1, "/stackFile/stack/card") into tNext
```

This results in *tPrev* containing:

/stackFile/stack/card[1]

And *tNext* containing:

/stackFile/stack/card[2]

Searching for a Node

To search for a node based on an attribute, use **revXMLMatchingNode**.

```
revXMLMatchingNode(treeID, startNode, childName, \\ attributeName, attributeValue, depth, [caseSens
itive])
```

See above for an explanation of *treeID*, *startNode* and *depth*.

The *childName* is the name of the children you want to include in the search. If you leave this blank all children are searched.

The *attributeName* is the name of the attribute you want to search.

attributeValue is the search term you want to match.

caseSensitive optionally specifies whether the search should be case sensitive. The default is false.

Using this function on our example XML file as follows:

```
put revXMLMatchingNode(106,"/", ,"name", "Hello", -1) into tMatch
```

This results in *tMatch* containing:

/stackFile/stack/card[1]/field

Retrieving an Outline of the Tree (or Portion Thereof)

To retrieve the contents of a specified node, use **revXMLTree**.

```
revXMLTree(treeID, startNode, nodeDelim, padding, includeChildCount, depth)
```

See above for an explanation of *treeID*, *startNode, includeChildCount* and *depth*.

The *nodeDelim* is the delimiter that separates each node in the tree. Use return to retrieve a list of nodes.

padding is the character to use to indent each level in the tree.

Using this function on our example XML file as follows:

```
put revXMLTree(106,"/",return,space,true,-1) into tTree
```

This results in *tTree* containing:

stackFile[1]

stack[1]

card[1]

field[1]

text[1]

htmlText[1]

card[2]

Retrieving the Tree as XML (or Portion Thereof)

To retrieve the tree as XML use **revXMLText**. Use this function to save the XML to a file after modifying it.

```
revXMLText(treeID, startNode, [formatTree])
```

See above for an explanation of *treeID* and *startNode*.

The *formatTree* specifies whether or not to format the returned tree with return and space characters to make it easier to read by a human.

Using this function on our example XML file as follows:

```
ask file "Save XML as:"
put revXMLText(106,"/",true) into URL ("file:" & it)
```

This results in the file the *user specifies* containing:

```
<stackFile>

<stack name="Example" rect="117,109,517,509">

<card>

<field name="Hello" rect="100,100,200,125">

<text>Hello World!</text>

<htmlText>&lt;&gt;Hello World&lt;/p&gt;</htmlText>

</field>

</card>

</stack>

</stackFile>
```

Validating against a DTD

To check the syntax of an XML file conforms to a DTD use **revXMLValidateDTD**. For more information on this function, see the *LiveCode Dictionary*.

Listing all XML Trees in Memory

To generate a list of all XML trees in memory, use **revXMLTrees**. For more information on this function, see the *LiveCode Dictionary*.

Removing an XML Tree from Memory

To remove an XML tree from memory, use **revXMLDeleteTree**. To remove all XML trees from memory, use **revXMLDeleteAllTrees**. Both functions take a single parameter – the ID of the tree to be deleted. You should delete a tree when you have stopped using it. For more information on these functions, see the *LiveCode Dictionary*.

Once an XML tree has been removed from memory, there is no way to get it back. Use the revXMLText function to retrieve the contents of the entire tree and save it first.

The XML Library: Editing XML

This section discusses how to edit XML trees. Before reading this section you should read the section above on loading, displaying and unloading XML.

Adding a new Child Node

To add a new node use the **revXMLAddNode** command.

```
revXMLAddNode treeID, parentNode, nodeName, nodeContents, [location]
```

See above for an explanation of *treeID*.

The *parentNode* is the name of the node you want to add the child to.

The *nodeName* is the name of the new node to create.

nodeContents is the contents of the new node.

location - optionally specify "before" to place the new child at the start of the child nodes.

Use this function to add a button to our example XML file as follows:

```
revXMLAddNode 1, "/stackFile/stack/card/", "button", ""
```

This results in our tree containing a new button:

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- This is a comment. -->

<stackFile>

<stack name="Example" rect="117,109,517,509">

<card>

<field name="Hello" rect="100,100,200,125">

<text>Hello World!</text>

<htmlText>&lt;p&gt;Hello World&lt;/p&gt;</htmlText>

</field>

<button></button>

</card>

</stack>

</stackFile>
```

To create another node at the same level as another node, use the **revXMLInsertNode** command instead.

Appending XML to a tree

To add a new node use the **revXMLAppend** command.

```
revXMLAppend treeID, parentNode, newXML
```

See above for an explanation of *treeID* and *parentNode*.

The *newXML* is XML you want to append to the tree.

Moving, Copying or Deleting a Node

To move a node use the **revXMLMoveNode** command.

```
revXMLMoveNode treeID, sourceNode, destinationNode [, location] [, relationship]
```

See above for an explanation of *treeID*.

The *sourceNode* is the path to the node you want to move.

The *destinationNode* is the path to the node you to move to.

The *location* specifies where the node should be moved to in the list of siblings – it can be either "before" or "after".

The *relationship* allows you to specify whether to place the node alongside the destination as a sibling or below the destination as a child.

To copy a node use **revXMLCopyNode**.

To delete a node use **revXMLDeleteNode**.

Putting data into a Node

To put data into a node use the **revXMLPutIntoNode** command.

```
revXMLPutIntoNode treeID,node,newContents
```

See above for an explanation of *treeID* and *node*.

The *newContents* is the text that the new node will contain.

Setting an Attribute

To set an attribute use the **revXMLSetAttribute** command.

```
revXMLSetAttribute treeID,node,attributeName,newValue
```

See above for an explanation of *treeID* and *node*.

The *attributeName* is the name of the attribute you want to set the attribute on.

The *newValue* is the value to set for the attribute.

Using this function to add a "showBorder" property to our field:

```
revXMLSetAttribute 1, "/stackFile/stack/card/button", "showBorder","true"
```

The field tag in our tree now looks like this:

```
<field name="Hello" rect="100,100,200,125" showBorder="true">
```

Adding a DTD

To add a DTD to the tree, use the **revXMLAddDTD** command.

```
revXMLAddDTD treeID,DTDText
```

See above for an explanation of *treeID*.

The *DTDText* is the text of the DTD to add.

Sorting

Sorting data is a common and fundamental operation. Sorting allows you to display data in a user-friendly fashion or code a number of algorithms. LiveCode's intuitive sort features give you the power and flexibility to perform any kind of sorting you may require.

The Sort Container Command: Overview

To sort data, use the **sort container** command.

```
sort [{lines | items} of] container [direction] [sortType] [by sortKey]
```

The *container* is a field, button, or variable, or the message box.

The *direction* is either ascending or descending. If you don't specify a direction, the sort is ascending.

The *sortType* is one of text, numeric, or dateTime. If you don't specify a sortType, the sort is by text.

The *sortKey* is an expression that evaluates to a value for each line or item in the container. If the *sortKey*contains a chunk expression, the keyword *each* indicates that the chunk expression is evaluated for *each* line or item. If you don't specify a sortKey, the entire line (or item) is used as the sortKey.

The following example sorts the *lines* of a variable alphabetically:

```
sort lines of field "sample text" ascending text
sort lines of tText descending text
```

The following example sorts a collection of *items* numerically:

```
sort items of field "sample csv" ascending numeric
sort items of tItems descending numeric
```

The Sort Container Command: Using Sort Keys

The *sortKey* syntax allows you to sort each line or item based on the results of an evaluation performed on each line or item.

To sort the lines of a container by a specific item in each line:

```
sort lines of tContainer by the first item of each
sort lines of tContainer by item 3 of each
```

The *sortKey* expression will only be evaluated once for every element that is to be sorted. This syntax allows a variety of more complex sort operations to be performed.

The following example will extract the minimum and maximum integers present in a list:

```
set the itemDelimiter to "."
sort lines of fld 1 numeric by char 2 to -1 of the first item of each
put char 2 to -1 of the first item \
    of the first line of fld 1 into tMinimumInteger
put char 2 to -1 of the first item \
    of the last line of fld 1 into tMaximumInteger
```

Original list:	Result:
F54.mov	tMinimumInteger is 3
M27.mov	tMaximumInteger is 54
M7.mov	
F3.mov	

Figure 57 – Results of sort command using sort key

The Sort Container Command: Sorting Randomly

To sort randomly, use the **random** function to generate a random number as the *sortKey* for each line or item, instead of evaluating the line or item's contents. For example:

```
put the number of lines of tExampleList into tElementCount
sort lines of tExampleList ascending numeric by random(tElementCount)
```

The Sort Container Command: Stable Sorts – Sorting on Multiple Keys

To sort a list by multiple criteria you can sort multiple times. This is because LiveCode uses a stable sort, meaning that if two items have the same sort key their relative order in the output will not change. To perform a stable sort, start with the least significant or important criteria and work up to the most important or significant. For example:

```
sort lines of fld 1 ascending numeric by item 2 of each
sort lines of fld 1 ascending text by the first item of each
```

Original list:	Result:
Oliver,1.54	Elanor,5.67
Elanor,5.67	Elanor,6.3
Marcus,8.99	Marcus,8.99
Elanor,6.34	Oliver,1.54
Oliver,8.99	Oliver,8.99
Tim,3.44	Tim,3.44

Figure 58 – Results of sorting multiple items

> **Tip:** If you have a large data set and want to improve performance by only performing a single sort, you can construct a sort key that gives the appropriate ordering. In this example a good way to do that is to use the **format** function to construct a fixed length string, one element per sort:

```
sort lines of fld 1 ascending text by \\
format("%-16s%08.2f", item 1 of each, item 2 of each)
```

This formats each individual line similar to the following:

```
Oliver 00001.54
Elanor 00005.67
```

These lines now sort the required way as if the first field (the name) ties, the order is determined by the second field – due to the use of padding characters making all the fields the same size.

Sorting Cards

To sort cards, use the **sort** command.

```
sort [marked] cards [of stack] [direction] [sortType] by sortKey
```

The *stack* is a reference to any open stack. If you don't specify a *stack*, the cards of the current stack are sorted.

The *direction* is either ascending or descending. If you don't specify a *direction*, the sort is ascending.

The *sortType* is one of text, international, numeric, or dateTime. If you don't specify a *sortType*, the sort is by text.

The *sortKey* is an expression that evaluates to a value for each card in the stack. Any object references within the *sortKey* are treated as pertaining to each card being evaluated, so for example, a reference to a field is evaluated according to that field's contents on each card. Typically the sort command is used with *background* fields that have their *sharedText* property set to false so that they contain a different value on each card.

For example to sort cards by the contents of the last name field on each:

```
sort cards by field "Last Name"
```

To sort cards by the numeric value in a ZIP Code:

```
sort cards numeric by field "ZIP code"
```

> **Tip:** To sort cards by a custom expression that performs a calculation, you can create a custom function:

```
sort cards by myFunction() -- uses function below

function myFunction
    put the number of buttons of this card into tValue
    -- perform any calculation on tValue here
    return tValue
    -- sort will use this value
end myFunction
```